W3School Memcached 教程

wizardforcel

Published with GitBook

目錄

W3School Memcached 教程

作者：W3School

来源：Memcached 教程

Memcached 入门

Memcached 简介

Memcached是一个自由开源的，高性能，分布式内存对象缓存系统。

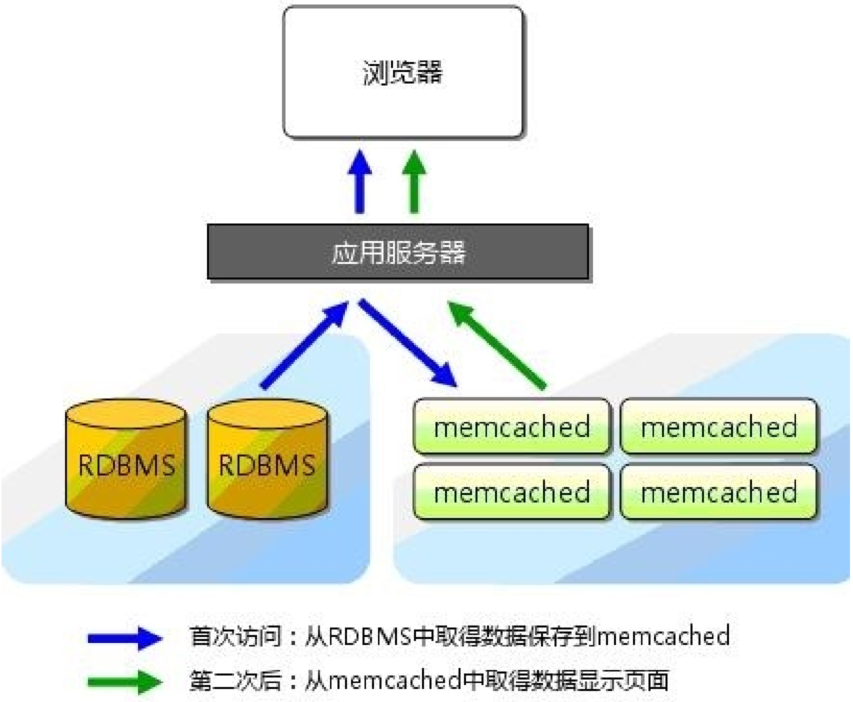
Memcached是以LiveJournal旗下Danga Interactive公司的Brad Fitzpatric为首开发的一款软件。现在已成为mixi、hatena、Facebook、Vox、LiveJournal等众多服务中提高Web应用扩展性的重要因素。

Memcached是一种基于内存的key-value存储，用来存储小块的任意数据（字符串、对象）。这些数据可以是数据库调用、API调用或者是页面渲染的结果。

Memcached简洁而强大。它的简洁设计便于快速开发，减轻开发难度，解决了大数据量缓存的很多问题。它的API兼容大部分流行的开发语言。

本质上，它是一个简洁的key-value存储系统。

一般的使用目的是，通过缓存数据库查询结果，减少数据库访问次数，以提高动态Web应用的速度、提高可扩展性。



Memcached 官网：http://memcached.org/。

特征

memcached作为高速运行的分布式缓存服务器，具有以下的特点。

- 协议简单
- 基于libevent的事件处理
- 内置内存存储方式

- memcached不互相通信的分布式

支持的语言

许多语言都实现了连接memcached的客户端，其中以Perl、PHP为主。仅仅memcached网站上列出的有：

- Perl
- PHP
- Python
- Ruby
- C#
- C/C++
- Lua
- 等等

Memcached 用户

- LiveJournal
- Wikipedia
- Flickr
- Bebo
- Twitter
- Typepad
- Yellowbot
- Youtube
- WordPress.com
- Craigslist
- Mixi

Memcached 安装

Memcached 支持许多平台：Linux、FreeBSD、Solaris、Mac OS，也可以安装在Windows上。

Linux系统安装memcached，首先要先安装libevent库。

```
sudo apt-get install libevent libevent-deve          自动下载安装（Ub
yum install libevent libevent-deve                      自动下载安装
```

安装 Memcached

自动安装

Ubuntu/Debian

```
sudo apt-get install memcached
```

Redhat/Fedora/Centos

```
yum install memcached
```

FreeBSD

```
portmaster databases/memcached
```

源代码安装

从其官方网站（http://memcached.org）下载memcached最新版本。

`wget http://memcached.org/latest`	下载最新版本
`tar -zxvf memcached-1.x.x.tar.gz`	解压源码
`cd memcached-1.x.x`	进入目录
`./configure --prefix=/usr/local/memcached`	配置
`make && make test`	编译
`sudo make install`	安装

Memcached 运行

Memcached命令的运行：

```
$ /usr/local/memcached/bin/memcached -h
```

注意：如果使用自动安装 memcached 命令位于 **/usr/local/bin/memcached**。

启动选项：

- -d是启动一个守护进程；
- -m是分配给Memcache使用的内存数量，单位是MB；
- -u是运行Memcache的用户；
- -l是监听的服务器IP地址，可以有多个地址；
- -p是设置Memcache监听的端口，，最好是1024以上的端口；
- -c是最大运行的并发连接数，默认是1024；
- -P是设置保存Memcache的pid文件。

（1）作为前台程序运行：

从终端输入以下命令，启动memcached:

```
/usr/local/memcached/bin/memcached -p 11211 -m 64m -vv

slab class   1: chunk size     88 perslab 11915

slab class   2: chunk size    112 perslab  9362

slab class   3: chunk size    144 perslab  7281

中间省略

slab class  38: chunk size 391224 perslab     2

slab class  39: chunk size 489032 perslab     2

<23 server listening

<24 send buffer was 110592, now 268435456

<24 server listening (udp)

<24 server listening (udp)

<24 server listening (udp)

<24 server listening (udp)
```

这里显示了调试信息。这样就在前台启动了memcached，监听TCP端口11211，最大内存使用量为64M。调试信息的内容大部分是关于存储的信息。

（2）作为后台服务程序运行：

```
# /usr/local/memcached/bin/memcached -p 11211 -m 64m -d
```

或者

```
/usr/local/memcached/bin/memcached -d -m 64M -u root -l 192.168.0.2
```

Memcached 连接

我们可以通过 telnet 命令并指定主机ip和端口来连接 Memcached 服务。

语法

```
telnet HOST PORT
```

命令中的 **HOST** 和 **PORT** 为运行 Memcached 服务的 IP 和 端口。

实例

以下实例演示了如何连接到 Memcached 服务并执行简单的 set 和 get 命令。

本实例的 Memcached 服务运行的主机为 127.0.0.1（本机）、端口为 11211。

```
telnet 127.0.0.1 11211
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
set foo 0 0 3                                                        保
bar                                                                  数
STORED                                                               结
get foo                                                              取
VALUE foo 0 3                                                        数
bar                                                                  数
END                                                                  结
quit                                                                 退
```

Memcached 存储命令

Memcached set 命令

Memcached set 命令用于将 **value(数据值)** 存储在指定的 **key(键)** 中。

如果set的key已经存在，该命令可以更新该key所对应的原来的数据，也就是实现更新的作用。

语法：

set 命令的基本语法格式如下：

```
set key flags exptime bytes [noreply]
value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息。
- **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
- **bytes**：在缓存中存储的字节数
- **noreply**（可选）： 该参数告知服务器不需要返回数据
- **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

以下实例中我们设置：

- key → runoob
- flag → 0
- exptime → 900 (以秒为单位)
- bytes → 9 (数据存储的字节数)
- value → memcached

```
set runoob 0 900 9
memcached
STORED

get runoob
VALUE runoob 0 9
memcached

END
```

输出

如果数据设置成功，则输出：

```
STORED
```

输出信息说明：

- **STORED**：保存成功后输出。
- **ERROR**：在保持失败后输出。

Memcached add 命令

Memcached add 命令用于将 **value(数据值)** 存储在指定的 **key(键)** 中。

如果 add 的 key 已经存在，则不会更新数据，之前的值将仍然保持相同，并且您将获得响应 **NOT_STORED**。

语法：

add 命令的基本语法格式如下：

```
add key flags exptime bytes [noreply]
value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息。
- **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
- **bytes**：在缓存中存储的字节数
- **noreply**（可选）： 该参数告知服务器不需要返回数据
- **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

以下实例中我们设置：

- key → new_key
- flag → 0
- exptime → 900 (以秒为单位)
- bytes → 10 (数据存储的字节数)
- value → data_value

```
add new_key 0 900 10
data_value
STORED
get new_key
VALUE new_key 0 10
data_value
END
```

输出

如果数据添加成功，则输出：

```
STORED
```

输出信息说明：

- **STORED**：保存成功后输出。
- **NOT_STORED** ：在保持失败后输出。

Memcached replace 命令

Memcached replace 命令用于替换已存在的 **key(键)** 的 **value(数据值)**。

如果 key 不存在，则替换失败，并且您将获得响应 **NOT_STORED**。

语法：

replace 命令的基本语法格式如下：

```
replace key flags exptime bytes [noreply]
value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息。
- **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
- **bytes**：在缓存中存储的字节数
- **noreply**（可选）： 该参数告知服务器不需要返回数据
- **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

以下实例中我们设置：

- key → mykey
- flag → 0
- exptime → 900 (以秒为单位)
- bytes → 10 (数据存储的字节数)
- value → data_value

以下实例中我们使用的键位 'mykey' 并存储对应的值 data_value。执行后我们替换相同的 key 的值为 'some_other_value'。

```
add mykey 0 900 10
data_value
STORED
get mykey
VALUE mykey 0 10
data_value
END
replace mykey 0 900 16
some_other_value
get mykey
VALUE mykey 0 16
some_other_value
END
```

输出

如果数据添加成功，则输出：

```
STORED
```

输出信息说明：

- **STORED**：保存成功后输出。
- **NOT_STORED**：执行替换失败后输出。

Memcached append 命令

Memcached append 命令用于向已存在 **key(键)** 的 **value(数据值)** 后面追加数据 。

语法：

append 命令的基本语法格式如下：

```
append key flags exptime bytes [noreply]
value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息 。
- **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
- **bytes**：在缓存中存储的字节数
- **noreply**（可选）： 该参数告知服务器不需要返回数据
- **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

实例如下：

- 首先我们在 Memcached 中存储一个键 runoob，其值为 memcached。
- 然后，我们使用 get 命令检索该值。
- 然后，我们使用 **append** 命令在键为 runoob 的值后面追加 "redis"。
- 最后，我们再使用 get 命令检索该值。

```
set runoob 0 900 9
memcached
STORED
get runoob
VALUE runoob 0 14
memcached
END
append runoob 0 900 5
redis
STORED
get runoob
VALUE runoob 0 14
memcachedredis
END
```

输出

如果数据添加成功，则输出：

```
STORED
```

输出信息说明：

- **STORED**：保存成功后输出。
- **NOT_STORED**：该键在 Memcached 上不存在。
- **CLIENT_ERROR**：执行错误。

Memcached prepend 命令

Memcached prepend 命令用于向已存在 **key(键)** 的 **value(数据值)** 前面追加数据
。

语法：

prepend 命令的基本语法格式如下：

```
prepend key flags exptime bytes [noreply]
value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息
  。
- **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
- **bytes**：在缓存中存储的字节数
- **noreply**（可选）： 该参数告知服务器不需要返回数据
- **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的
  value）

实例

实例如下：

- 首先我们在 Memcached 中存储一个键 runoob，其值为 memcached。
- 然后，我们使用 get 命令检索该值。
- 然后，我们使用 **prepend** 命令在键为 runoob 的值后面追加 "redis"。
- 最后，我们再使用 get 命令检索该值。

```
set runoob 0 900 9
memcached
STORED
get runoob
VALUE runoob 0 14
memcached
END
prepend runoob 0 900 5
redis
STORED
get runoob
VALUE runoob 0 14
redismemcached
END
```

输出

如果数据添加成功，则输出：

```
STORED
```

输出信息说明：

- **STORED**：保存成功后输出。
- **NOT_STORED**：该键在 Memcached 上不存在。
- **CLIENT_ERROR**：执行错误。

Memcached CAS 命令

Memcached CAS（Check-And-Set 或 Compare-And-Swap） 命令用于执行一个"检查并设置"的操作

它仅在当前客户端最后一次取值后，该key 对应的值没有被其他客户端修改的情况下， 才能够将值写入。

检查是通过cas_token参数进行的， 这个参数是Memcach指定给已经存在的元素的一个唯一的64位值。

语法：

CAS 命令的基本语法格式如下：

```
cas key flags exptime bytes unique_cas_token [noreply]
value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息。
- **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
- **bytes**：在缓存中存储的字节数
- **unique_cas_token**通过 gets 命令获取的一个唯一的64位值。
- **noreply**（可选）： 该参数告知服务器不需要返回数据
- **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

要在 Memcached 上使用 CAS 命令，你需要从 Memcached 服务商通过 gets 命令获取令牌（token）。

gets 命令的功能类似于基本的 get 命令。两个命令之间的差异在于，gets 返回的信息稍微多一些：64 位的整型值非常像名称/值对的 "版本" 标识符。

实例步骤如下：

- 如果没有设置唯一令牌，则 CAS 命令执行错误。
- 如果键 key 不存在，执行失败。
- 添加键值对。
- 通过 gets 命令获取唯一令牌。
- 使用 cas 命令更新数据

- 使用 get 命令查看数据是否更新

```
cas tp 0 900 9
ERROR             <? 缺少 token

cas tp 0 900 9 2
memcached
NOT_FOUND         <? 键 tp 不存在

set tp 0 900 9
memcached
STORED

gets tp
VALUE tp 0 9 1
memcached
END

cas tp 0 900 5 1
redis
STORED

get tp
VALUE tp 0 5
redis
END
```

输出

如果数据添加成功，则输出：

```
STORED
```

输出信息说明：

- **STORED**：保存成功后输出。
- **ERROR**：保存出错或语法错误。
- **EXISTS**：在最后一次取值后另外一个用户也在更新该数据。
- **NOT_FOUND**：Memcached 服务上不存在该键值。

Memcached 查找命令

Memcached get 命令

Memcached get 命令获取存储在 **key(键)** 中的 **value(数据值)** ，如果 key 不存在，则返回空。

语法：

get 命令的基本语法格式如下：

```
get key
```

多个 key 使用空格隔开，如下:

```
get key1 key2 key3
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。

实例

在以下实例中，我们使用 runoob 作为 key，过期时间设置为 900 秒。

```
set runoob 0 900 9
memcached
STORED
get runoob
VALUE runoob 0 9
memcached
END
```

Memcached gets 命令

Memcached gets 命令获取带有 CAS 令牌存 的 **value(数据值)**，如果 key 不存在，则返回空。

语法：

gets 命令的基本语法格式如下：

```
gets key
```

多个 key 使用空格隔开，如下:

```
gets key1 key2 key3
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。

实例

在以下实例中，我们使用 runoob 作为 key，过期时间设置为 900 秒。

```
set runoob 0 900 9
memcached
STORED
gets runoob
VALUE runoob 0 9 1
memcached
END
```

在 使用 gets 命令的输出结果中，在最后一列的数字 1 代表了 key 为 runoob 的 CAS 令牌。

Memcached delete 命令

Memcached delete 命令用于删除已存在的 key(键)。

语法：

delete 命令的基本语法格式如下：

```
delete key [noreply]
```

多个 key 使用空格隔开，如下:

```
gets key1 key2 key3
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **noreply**（可选）： 该参数告知服务器不需要返回数据

实例

在以下实例中，我们使用 runoob 作为 key，过期时间设置为 900 秒。之后我们使用 delete 命令删除该 key。

```
set runoob 0 900 9
memcached
STORED
get runoob
VALUE runoob 0 9
memcached
END
delete runoob
DELETED
get runoob
END
delete runoob
NOT_FOUND
```

输出

输出信息说明：

- **DELETED**：删除成功。

- **ERROR**：语法错误或删除失败。
- **NOT_FOUND**：key 不存在。

Memcached incr 与 decr 命令

Memcached incr 与 decr 命令用于对已存在的 key(键) 的数字值进行自增或自减操作。

incr 与 decr 命令操作的数据必须是十进制的32位无符号整数。

如果 key 不存在返回 **NOT_FOUND**，如果键的值不为数字，则返回 **CLIENT_ERROR**，其他错误返回 **ERROR**。

incr 命令

语法：

incr 命令的基本语法格式如下：

```
incr key increment_value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **increment_value**： 增加的数值。

实例

在以下实例中，我们使用 visitors 作为 key，初始值为 10，之后进行加 5 操作。

```
set visitors 0 900 2
10
STORED
get visitors
VALUE visitors 0 2
10
END
incr visitors 5
15
get visitors
VALUE visitors 0 2
15
END
```

输出

输出信息说明：

- **NOT_FOUND**：key 不存在。
- **CLIENT_ERROR**：自增值不是对象。
- **ERROR**其他错误，如语法错误等。

decr 命令

decr 命令的基本语法格式如下：

```
decr key decrement_value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **decrement_value**： 减少的数值。

实例

```
set visitors 0 900 2
10
STORED
get visitors
VALUE visitors 0 2
10
END
decr visitors 5
5
get visitors
VALUE visitors 0 1
5
END
```

在以下实例中，我们使用 visitors 作为 key，初始值为 10，之后进行减 5 操作。

输出

输出信息说明：

- **NOT_FOUND**：key 不存在。
- **CLIENT_ERROR**：自增值不是对象。
- **ERROR**其他错误，如语法错误等。

Memcached 统计命令

Memcached stats 命令

Memcached stats 命令用于返回统计信息例如 PID(进程号)、版本号、连接数等。

语法：

stats 命令的基本语法格式如下：

```
stats
```

实例

在以下实例中，我们使用了 stats 命令来输出 Memcached 服务信息。

```
stats
STAT pid 1162
STAT uptime 5022
STAT time 1415208270
STAT version 1.4.14
STAT libevent 2.0.19-stable
STAT pointer_size 64
STAT rusage_user 0.096006
STAT rusage_system 0.152009
STAT curr_connections 5
STAT total_connections 6
STAT connection_structures 6
STAT reserved_fds 20
STAT cmd_get 6
STAT cmd_set 4
STAT cmd_flush 0
STAT cmd_touch 0
STAT get_hits 4
STAT get_misses 2
STAT delete_misses 1
STAT delete_hits 1
STAT incr_misses 2
STAT incr_hits 1
STAT decr_misses 0
STAT decr_hits 1
STAT cas_misses 0
STAT cas_hits 0
STAT cas_badval 0
STAT touch_hits 0
STAT touch_misses 0
STAT auth_cmds 0
STAT auth_errors 0
STAT bytes_read 262
STAT bytes_written 313
STAT limit_maxbytes 67108864
STAT accepting_conns 1
STAT listen_disabled_num 0
STAT threads 4
STAT conn_yields 0
STAT hash_power_level 16
STAT hash_bytes 524288
STAT hash_is_expanding 0
STAT expired_unfetched 1
STAT evicted_unfetched 0
STAT bytes 142
STAT curr_items 2
STAT total_items 6
STAT evictions 0
STAT reclaimed 1
END
```

这里显示了很多状态信息，下边详细解释每个状态项：

- **pid**： memcache服务器进程ID
- **uptime**：服务器已运行秒数
- **time**：服务器当前Unix时间戳
- **version**：memcache版本
- **pointer_size**：操作系统指针大小
- **rusage_user**：进程累计用户时间
- **rusage_system**：进程累计系统时间
- **curr_connections**：当前连接数量
- **total_connections**：Memcached运行以来连接总数
- **connection_structures**：Memcached分配的连接结构数量
- **cmd_get**：get命令请求次数
- **cmd_set**：set命令请求次数
- **cmd_flush**：flush命令请求次数
- **get_hits**：get命令命中次数
- **get_misses**：get命令未命中次数
- **delete_misses**：delete命令未命中次数
- **delete_hits**：delete命令命中次数
- **incr_misses**：incr命令未命中次数
- **incr_hits**：incr命令命中次数
- **decr_misses**：decr命令未命中次数
- **decr_hits**：decr命令命中次数
- **cas_misses**：cas命令未命中次数
- **cas_hits**：cas命令命中次数
- **cas_badval**：使用擦拭次数
- **auth_cmds**：认证命令处理的次数
- **auth_errors**：认证失败数目
- **bytes_read**：读取总字节数
- **bytes_written**：发送总字节数
- **limit_maxbytes**：分配的内存总大小（字节）
- **accepting_conns**：服务器是否达到过最大连接（0/1）
- **listen_disabled_num**：失效的监听数
- **threads**：当前线程数
- **conn_yields**：连接操作主动放弃数目
- **bytes**：当前存储占用的字节数
- **curr_items**：当前存储的数据总数
- **total_items**：启动以来存储的数据总数
- **evictions**：LRU释放的对象数目
- **reclaimed**：已过期的数据条目来存储新数据的数目

Memcached stats items 命令

Memcached stats items 命令用于显示各个 slab 中 item 的数目和存储时长(最后一次访问距离现在的秒数)。

语法：

stats items 命令的基本语法格式如下：

```
stats items
```

实例

```
stats items
STAT items:1:number 1
STAT items:1:age 7
STAT items:1:evicted 0
STAT items:1:evicted_nonzero 0
STAT items:1:evicted_time 0
STAT items:1:outofmemory 0
STAT items:1:tailrepairs 0
STAT items:1:reclaimed 0
STAT items:1:expired_unfetched 0
STAT items:1:evicted_unfetched 0
END
```

Memcached stats slabs 命令

Memcached stats slabs 命令用于显示各个slab的信息，包括chunk的大小、数目、使用情况等。

语法：

stats slabs 命令的基本语法格式如下：

```
stats slabs
```

实例

```
stats slabs
STAT 1:chunk_size 96
STAT 1:chunks_per_page 10922
STAT 1:total_pages 1
STAT 1:total_chunks 10922
STAT 1:used_chunks 1
STAT 1:free_chunks 10921
STAT 1:free_chunks_end 0
STAT 1:mem_requested 71
STAT 1:get_hits 0
STAT 1:cmd_set 1
STAT 1:delete_hits 0
STAT 1:incr_hits 0
STAT 1:decr_hits 0
STAT 1:cas_hits 0
STAT 1:cas_badval 0
STAT 1:touch_hits 0
STAT active_slabs 1
STAT total_malloced 1048512
END
```

Memcached stats sizes 命令

Memcached stats sizes 命令用于显示所有item的大小和个数。

该信息返回两列，第一列是 item 的大小，第二列是 item 的个数。

语法：

stats sizes 命令的基本语法格式如下：

```
stats sizes
```

实例

```
stats sizes
STAT 96 1
END
```

Memcached flush_all 命令

Memcached flush_all 命令用于用于清理缓存中的所有 **key=>value(键=>值)** 对。

该命令提供了一个可选参数 **time**，用于在制定的时间后执行清理缓存操作。

语法：

flush_all 命令的基本语法格式如下：

```
flush_all [time] [noreply]
```

实例

清理缓存:

```
set runoob 0 900 9
memcached
STORED
get runoob
VALUE runoob 0 9
memcached
END
flush_all
OK
get runoob
END
```

Memcached 实例

Java 连接 Memcached 服务

使用 Java 程序连接 Memcached，需要在你的 classpath 中添加 Memcached jar 包。

本站 jar 包下载地址：spymemcached-2.10.3.jar。Google Code jar 包下载地址：spymemcached-2.10.3.jar（需要翻墙）。

以下程序假定 Memcached 服务的主机为 127.0.0.1，端口为 11211。

连接实例

Java 连接 Memcached

```
import net.spy.memcached.MemcachedClient;
import java.net.*;

public class MemcachedJava {
   public static void main(String[] args) {
      try{
         // 本地连接 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex){
         System.out.println( ex.getMessage() );
      }
   }
}
```

该程序中我们使用 InetSocketAddress 连接 IP 为 127.0.0.1 端口 为 11211 的 memcached 服务。

执行以上代码，如果连接成功会输出以下信息：

```
Connection to server successful.
```

set 操作实例

以下使用 java.util.concurrent.Future 来存储数据

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{
         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketA
         System.out.println("Connection to server sucessful.");

         // 存储数据
         Future fo = mcc.set("runoob", 900, "Free Education");

         // 查看存储状态
         System.out.println("set status:" + fo.get());

         // 输出值
         System.out.println("runoob value in cache - " + mcc.get("
         
         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex){
         System.out.println( ex.getMessage() );
      }
   }
}
```

执行程序，输出结果为：

```
Connection to server successful.
set status:true
runoob value in cache - Free Education
```

add 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 添加数据
         Future fo = mcc.set("runoob", 900, "Free Education");

         // 打印状态
         System.out.println("set status:" + fo.get());

         // 输出
         System.out.println("runoob value in cache - " + mcc.get("r

         // 添加
         Future fo = mcc.add("runoob", 900, "memcached");

         // 打印状态
         System.out.println("add status:" + fo.get());

         // 添加新key
         fo = mcc.add("codingground", 900, "All Free Compilers");

         // 打印状态
         System.out.println("add status:" + fo.get());

         // 输出
         System.out.println("codingground value in cache - " + mcc.

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex){
         System.out.println(ex.getMessage());
      }
   }
}
```

replace 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try {
         //连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 添加第一个 key=》value 对
         Future fo = mcc.set("runoob", 900, "Free Education");

         // 输出执行 add 方法后的状态
         System.out.println("add status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("r

         // 添加新的 key
         fo = mcc.replace("runoob", 900, "Largest Tutorials' Librar

         // 输出执行 set 方法后的状态
         System.out.println("replace status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("r

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex){
         System.out.println( ex.getMessage() );
      }
   }
}
```

append 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 添加数据
         Future fo = mcc.set("runoob", 900, "Free Education");

         // 输出执行 set 方法后的状态
         System.out.println("set status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("r

         // 对存在的key进行数据添加操作
         Future fo = mcc.append("runoob", 900, " for All");

         // 输出执行 set 方法后的状态
         System.out.println("append status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("c

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex)
         System.out.println(ex.getMessage());
   }
}
```

prepend 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 添加数据
         Future fo = mcc.set("runoob", 900, "Education for All");

         // 输出执行 set 方法后的状态
         System.out.println("set status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("r

         // 对存在的key进行数据添加操作
         Future fo = mcc.prepend("runoob", 900, "Free ");

         // 输出执行 set 方法后的状态
         System.out.println("prepend status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("c

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex)
         System.out.println(ex.getMessage());
   }
}
```

CAS 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.CASValue;
import net.spy.memcached.CASResponse;
import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketA
         System.out.println("Connection to server sucessful.");

         // 添加数据
         Future fo = mcc.set("runoob", 900, "Free Education");

         // 输出执行 set 方法后的状态
         System.out.println("set status:" + fo.get());

         // 使用 get 方法获取数据
         System.out.println("runoob value in cache - " + mcc.get("r

         // 通过 gets 方法获取 CAS token（令牌）
         CASValue casValue = mcc.gets("runoob");

         // 输出 CAS token（令牌） 值
         System.out.println("CAS token - " + casValue);

         // 尝试使用cas方法来更新数据
         CASResponse casresp = mcc.cas("runoob", casValue.getCas(),

         // 输出 CAS 响应信息
         System.out.println("CAS Response - " + casresp);

         // 输出值
         System.out.println("runoob value in cache - " + mcc.get("r

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex)
         System.out.println(ex.getMessage());
   }
}
```

get 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 添加数据
         Future fo = mcc.set("runoob", 900, "Free Education");

         // 输出执行 set 方法后的状态
         System.out.println("set status:" + fo.get());

         // 使用 get 方法获取数据
         System.out.println("runoob value in cache - " + mcc.get("r

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex)
         System.out.println(ex.getMessage());
   }
}
```

gets 操作实例、CAS

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.CASValue;
import net.spy.memcached.CASResponse;
import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 添加数据
         Future fo = mcc.set("runoob", 900, "Free Education");

         // 输出执行 set 方法后的状态
         System.out.println("set status:" + fo.get());

         // 从缓存中获取键为 runoob 的值
         System.out.println("runoob value in cache - " + mcc.get("r

         // 通过 gets 方法获取 CAS token（令牌）
         CASValue casValue = mcc.gets("runoob");

         // 输出 CAS token（令牌） 值
         System.out.println("CAS value in cache - " + casValue);

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex)
         System.out.println(ex.getMessage());
   }
}
```

delete 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketA
         System.out.println("Connection to server sucessful.");

         // 添加数据
         Future fo = mcc.set("runoob", 900, "World's largest online

         // 输出执行 set 方法后的状态
         System.out.println("set status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("r

         // 对存在的key进行数据添加操作
         Future fo = mcc.delete("runoob");

         // 输出执行 delete 方法后的状态
         System.out.println("delete status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("c

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex)
         System.out.println(ex.getMessage());
   }
}
```

Incr/Decr 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 添加数字值
         Future fo = mcc.set("number", 900, "1000");

         // 输出执行 set 方法后的状态
         System.out.println("set status:" + fo.get());

         // 获取键对应的值
         System.out.println("value in cache - " + mcc.get("number")

         // 自增并输出
         System.out.println("value in cache after increment - " + m

         // 自减并输出
         System.out.println("value in cache after decrement - " + m

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex)
         System.out.println(ex.getMessage());
   }
}
```

PHP 连接 Memcached 服务

在前面章节中我们已经介绍了如何安装 Memcached 服务，接下来我们为大家介绍 PHP 如何使用 Memcached 服务。

PHP Memcache 扩展安装

PHP Memcache 扩展包下载地址：http://pecl.php.net/package/memcache，你可以下载最新稳定包(stable)。

```
wget http://pecl.php.net/get/memcache-2.2.7.tgz
tar -zxvf memcache-2.2.7.tgz
cd memcache-2.2.7
/usr/local/php/bin/phpize
./configure --with-php-config=/usr/local/php/bin/php-config
make && make install
```

注意：/usr/local/php/ 为php的安装路径，需要根据你安装的实际目录调整。

安装成功后会显示你的memcache.so扩展的位置，比如我的：

```
Installing shared extensions:     /usr/local/php/lib/php/extensions
```

最后我们需要把这个扩展添加到php中，打开你的php.ini文件在最后添加以下内容：

```
[Memcache]
extension_dir = "/usr/local/php/lib/php/extensions/no-debug-non-zts
extension = memcache.so
```

添加完后 重新启动php,我使用的是nginx+php-fpm进程所以命令如下：

```
kill -USR2 `cat /usr/local/php/var/run/php-fpm.pid`
```

如果是apache的使用以下命令:

```
/usr/local/apache2/bin/apachectl restart
```

检查安装结果

```
/usr/local/php/bin/php -m | grep memcache
```

安装成功会输出：memcache。

或者通过浏览器访问 phpinfo() 函数来查看，如下图：

memcache

memcache support	enabled
Active persistent connections	0
Version	2.2.7
Revision	$Revision: 327750 $

Directive	Local Value	Master Value
memcache.allow_failover	1	1
memcache.chunk_size	8192	8192
memcache.default_port	11211	11211
memcache.default_timeout_ms	1000	1000
memcache.hash_function	crc32	crc32
memcache.hash_strategy	standard	standard
memcache.max_failover_attempts	20	20

PHP 连接 Memcached

```
<?php
$memcache = new Memcache;             //创建一个memcache对象
$memcache->connect('localhost', 11211) or die ("Could not connect"
$memcache->set('key', 'test');        //设置一个变量到内存中，名称是key
$get_value = $memcache->get('key');   //从内存中取出key的值
echo $get_value;
?>
```

更多 PHP 操作 Memcached 请参阅：
http://php.net/manual/zh/book.memcache.php